Averaging and Statistics

Area Averaging

Area averaging is one of the most common data reduction procedures used in climate data analysis. The cdutil package has a powerful area averaging function. The averager() function provides a convenient way of averaging your data giving you control over the order of operations (i.e which dimensions are averaged over first) and also the weighting for the different axes. You can pass your own array of weights for each dimension or use the default (grid) weights or specify equal weighting.

Examples:

```
import cdms, cdutil

f = cdms.open('data_file_name')

# extract the variable 'var_name' from f
# and average over the dimension whose position is 1.
# Since no other options are specified,
# defaults kick in i.e weights='generate' (same as
# weights='weighted') and returned=0
result = cdutil.averager(f('var_name'), axis='1')

# Some ways of using the averager are shown below.

# A quick zonal mean calculation would be:
# (default weights option of 'generate' (or 'weighted') is implied)
V_zonal_ave = cdutil.averager(V, axis='x')

# If you want to average first over the x (longitude)
# dimension with area weighting and then over
# y (latitude) with equal weighting, then you would:
Vavg = cdutil.averager(V, axis='xy', weight=['generate','equal'])

# Similarly for equally weighted time averaging,
# where mywts is an array of shape (len(xaxis)) or shape(V):
cdutil.averager(V, axis='t', weight='equal')
cdutil.averager(V, axis='x', weight=mywts)


# myxwts is of shape len(xaxis),
# myywts is of shape len(yaxis)
cdutil.averager(V, axis='(lon)y', weight=[myxwts, myywts])

# V_wts is a Masked Variable of shape V
cdutil.averager(V, axis='xy', weight=V_wts)

# or the following will return the equally weighted sum
# over the x dimension, or
# is a good way to compute the area fraction that the
# data y that is non-missing

cdutil.averager(V, axis='x', weight='equal', action='sum')
ywt = cdutil.area_weights(y)
fractional_area= cdutil.averager(ywt, axis='xy',\
        weight=['equal','equal'],\
        action='sum')
```

Note: When averaging data with missing values, extra care needs to be taken. It is recommended that you use the default weight='generate' option. This uses cdutil.area_weights(V) to get the correct weights to pass to the averager.

```
cdutil.averager(V, axis='xy', weight='generate')

# The above is equivalent to:
V_wts = cdutil.area_weights(V)
result = cdutil.averager(V, axis='xy', weight=V_wts)
result = cdutil.averager(V, axis='xy', weight=cdutil.area_weights(V))
```

However, the area_weights function requires that the axis bounds are stored or can be calculated. In the case that such weights are not stored with the axis specifications (or the user desires to specify weights from another source), the use of combinewts option can produce the same results. In short, the following two are equivalent:

```
xavg_1 = averager(X, axis = 'xy', weights = area_weights(X))
xavg_2 = averager(X, axis = 'xy',\
        weights = ['weighted', 'weighted', 'weighted'],\
        combinewts=1)
```

Where X is a function of latitude, longitude and a third dimension such as time or level. In general, where the 'weighted' keyword appears above, it can be substituted with arrays of weights .

The following example will help you see the averager() function in context:

```
import cdms, cdutil
f = cdms.open('file_name')

# Using cdutil.domain to specify the NINO3 region
NINO3 = cdutil.domain(latitude=(-5.,5.),\ longitude=(210.,270.)))

# Extract the variable over the specified domain
nino3_area_exact = f('t', NINO3)

# Average first over the longitude axis
# (denoted by 'x') and then the latitude axis
# (denoted by 'y')
nino3_avg = cdutil.averager(nino3_area_exact, axis='xy')
```

Axis options can also be specified by name such as axis = '(depth)' or by index such as axis = '20' (note the numbers are enclosed in quotes). By default, the appropriate area weights are generated from the grid information and the result of the averaging is the area weighted value. More control over the weights used is available. It is possible to specify the weights used to average over the longitude and latitude axes seperately.

```
nino3avg2 = cdutil.averager(nino3_area_exact, axis='yx',weights=['generate','equal'])
```

In the above example, we averaged over the latitude axis first (using generated weights) and then over the longitude axis (using equal weights). The weights can be "equal" or "generate"(generates the weights for the grid information contained in the variable) or any array of numbers the user wishes to apply.

Generating Weights

For most averaging applications, the weights used are critical especially when there are missing data. The cdutil package provides a way of generating the weights using grid information that is tied to the variable. The averager function uses this to generate the weights when the default averaging weights option kicks in. This function is easily called for some variable 'x' in memory:

```
gen_weights = cdutil.area_weights(x)
```

The resultant gen_weights is in the same shape as the variable x and has the appropriate area weights set to missing values where data was missing in x.

Time Averaging

Averaging over time is a special problem in climate data analysis. The cdutil package pays special attention to this issue to make the extraction of time averages and climatologies simple. Apart from functions that enable easy computation of annual, seasonal and monthly averages and climatologies, one can also define seasons other than those already available and specify criteria for data availability and temporal distribution to suit individual needs.

Note: It is essential that the data have an appropriate axis designated as the "time" axis. In addition to this, the results depend on the time axis having correctly set "bounds". If "bounds" are not stored with the data in files, default "bounds" are generated by the data extraction steps in cdms. However, they are not always correct. The user must take care to verify that the bounds are set correctly.

The predefined time averaging periods are:

```
JAN, FEB, MAR,  ...., DEC       # months
DJF, MAM, JJA, SON              # seasons
YEAR                            # annual means
ANNUALCYCLE                     # monthly means for each month of the year
SEASONNALCYCLE                  # means for the 4 predefined seasons
```

Some simple examples of time averaging operations are shown here:

```
import cdutil

# To compute the DJF (december-January-February)
# climatology of a variable x
>>> djfclim = cdutil.DJF.climatology(x)

# The individual DJF seasons are extracted using
>>> djfs = cdutil.DJF(x)

# To extract DJF seasonal anomalies (from climatology)
>>> djf_anom = cdutil.DJF.departures(x)

# The monthly anomalies for x are computed by:
>>> x_anom = cdutil.ANNUALCYCLE.departures(x)
```

Creating Custom Seasons

You can even create your own "custom seasons" beyond the pre−defined seasons listed above. For example:

```
JJAS = cdutil.times.Seasons('JJAS')
```

Specifying the Time Period for Climatologies

So far we have seen the way to compute the means, climatologies, and anomalies for the entire length of the time−series. The typical application may require specified time intervals over which climatologies are computed and used in calculating departures. For example, to compute the DJF climatology for the time period 1979−1988 we would do the following:

```
import cdtime

start_time = cdtime.comptime(1979)
print 'start_time = ', start_time
end_time = cdtime.comptime(1989)
print 'end_time = ', end_time
```

Note that we created the time point 'end_time' at the begining of 89 so we can select all the time between 'start_time' and 'end_time' but not including 'end_time' by specifying the option 'co' − shorthand for 'c'losed at start_time and 'o'pen at end_time. For more details on different options available, refer to Climate Data Management System (cdms.pdf).

```
>>> djfclim = cdutil.DJF.climatology(x(time=(start_time, end_time, 'co')))
```

Now that we have our climatology over the desired period we can to compute anomalies over the full period relative to that climatology.

```
>>> djfdep2 = cdutil.DJF.departures(s, ref=djfclim)
```

Specifying Data Coverage Criteria

The real power of these functions is in the ability to specify minimum data coverage and to also be able to specify the distribution (both in the temporal sense) which are required for the averages to be computed. The default behaviour of the functions that compute seasonal averages, climatologies etc. is to require that a minimum of 50% of the data be present. Now let's say you like to extract DJF but without restricting it to 50% of the data being present. You would do:

```
djfs = cdutil.DJF(avg, criteriaarg=[0., None])
```

The above statement computes the DJF average with "criteriaarg" (passed as a list) which has 2 arguments.

- The first argument represents the minimum fraction of time that is required to compute the seasonal mean. So you can pass a fractional value between 0.0 and 1.0 (including both extremes) or even a representation such as 3.0/4.0 (in case you need at least 3 out of 4 months

of data in the case of the average JJAS we defined previously).
- The second argument in the criteriaarg is "None". This implies no "centroid function" is used. In other cases this argument represents the maximum value of the "centroid function".A value between 0 and 1 represents the spread of values across the mean time. The centroid value of 0.0 represents a full even distribution of data across the time interval. For example, if you are considering the DJF average, then if data is available for Dec, Jan and Feb months then the centroid is 0.0. On the other hand, the following criteria will "mask"(i.e ignore) a DJF season if there is only a december month with data (and therefore has a centroid value of 1.0). Therefore any seasons resulting in centroid values above 0.5 will result in missing values!

```
djfs = cdutil.DJF(avg, criteriaarg = [0., .5])
```

In the case of computing an annual mean, having data only in Jan and Dec months leads to a centroid value of 0 for the regular centroid, and the resulting annual mean for the year is biased toward the winter. In this situation, you should use a cyclical centroid where the circular nature of the year is recognised and the centroid is calculated accordingly. Here are some examples of typical usage:

1) Default behaviour is criteriaarg=[0.5, None]

```
annavg_1 = cdutil.YEAR(s_glavg)
```

2) Criteria to compute annual average for any number of months.

```
annavg_2 = cdutil.YEAR(s_glavg, criteriaarg = [0.,None])
```

3) Criteria for computing annual average based on the minimum number of months (8 out of 12).

```
annavg_3 = cdutil.YEAR(s_glavg, criteriaarg = [8./12., None])
```

4) Same criteria as in 3, but we account for the fact that a year is cyclical i.e Dec and Jan are adjacent months. So the centroid is computed over a circle where Dec and Jan are contiguous.

```
annavg_4 = cdutil.YEAR(s_glavg, criteriaarg = [8./12., 0.1, 'cyclical'])
```

So far we have the annual means calculated using various criteria. Now if we wish to compute the climatological annual mean, we can average the individual annual means. However, we can apply more criteria to the calculation of that annual mean climatology. Here we simply require 60% of the years to be present, and a criteria on the temporal distribution (i.e the centroid = 0.7) to make sure all of the annual means are not clustered at the end of the record.

```
annavg_clim = cdutil.YEAR.average(annavg_4,\ criteriaarg =[.6,.7])
```

The tutorial file times_tutorial.py has detailed examples of time averaging in action. Further documentation is available in the CDAT Utilities Reference Manual cdat_utilities.pdf.

Useful Statistical Functions

Commonly used statistical functions such as corrrelation, covariance, autocorrelation, autocovariance, laggedcorrelation, laggedcovariance, rms, variance, standard deviation, mean absolute difference, geometric mean, and linearregression have been implemented to allow easy computation of statistics. The statistics functions are implemented as part of the genutil package. These functions are implemented so as to not require the full variable information in MV. That is, these functions accept Numeric arrays. However, they also accept MV's so that the user can specify axes over which statistics are computed (to allow for spatial or temporal statistics etc.). The tutorial file statistics_tutorial.py shows some of the statistics functions in action.

Let us try an example where we want to look at a variable 'tas' from the NCEP reanalysis and compute some spatial statistics between data slices for time periods from 1960−1970 and 1980−1990:

```
import cdms
from genutil import statistics

f = cdms.open('tas.rnl_ncep.nc')
ncep1 = f('tas',time=('1960-1-1', '1970-1-1', 'co'))
ncep2 = f('tas',time=('1980-1-1', '1990-1-1', 'co'))

# We have the two time periods extracted.
# Now let us compute the correlation.
cor = statistics.correlation(ncep1, ncep2, axis='xy')

# We could compute the spatial correlation weighted by
# area. To accomplish this we can use the 'generate'
# option for weights.
wcor = statistics.correlation(ncep1, ncep2, weights='generate', axis='xy')
```

To compute the mean absolute difference between ncep1 and ncep2 defined in the previous example:

```
absd = statistics.meanabsdiff(ncep1, ncep2,axis='xy')
```

To compute the temporal rms difference between the two time periods:

```
rms = statistics.rms(ncep1, ncep2, axis='t')
```

In this example, we examine the default behaviour of the linearregression function:

```
Values = statistics.linearregression(y)
```

The returned "Values" is actually a tuple consisting of the slope and itercept. They can also be accessed as follows:

```
slope, intercept = statistics.linearregression(y)
```

If error estimates are also required, then:

```
Values, Errors = linearregression(y, error=1)
```

where "Values" and "Errors" are tuples containing answer for slope AND intercept. You can break them as follows. slope, intercept = Value and slope_error, intercept_error = Errors. i.e.

```
(slope, intercept), (slope_error, intercept_error) = linearregression(y, error=1)
```

WARNING: The following will not work.

```
slope, intercept, slo_error, int_error = linearregression(y, error=1)
```

To get the standard error non adjusted result for slope only, do the following:

```
slope, slope_error = linearregression(y, error=1, nointercept=1)
```

In the line below all the returned values are tuples.

```
Values,Errors,Pt1,Pt2,Pf1,Pf2 = linearregression(y, error=2,probability=1)
```

That means in the above statement is returning tuples ordered so: (slope, intercept), (slo_error, int_error), (pt1_slo, pt1_int), (pt2_slo, pt2_int), (pf1_slo, pf1_int), (pf2_slo, pf2_int).
If we want results returned for the intercept only, do the following:

```
intercept,intercept_error,pt1,pt2,pf1,pf2 =\
        linearregression(y,error=2,probability=1,noslope=1)
```